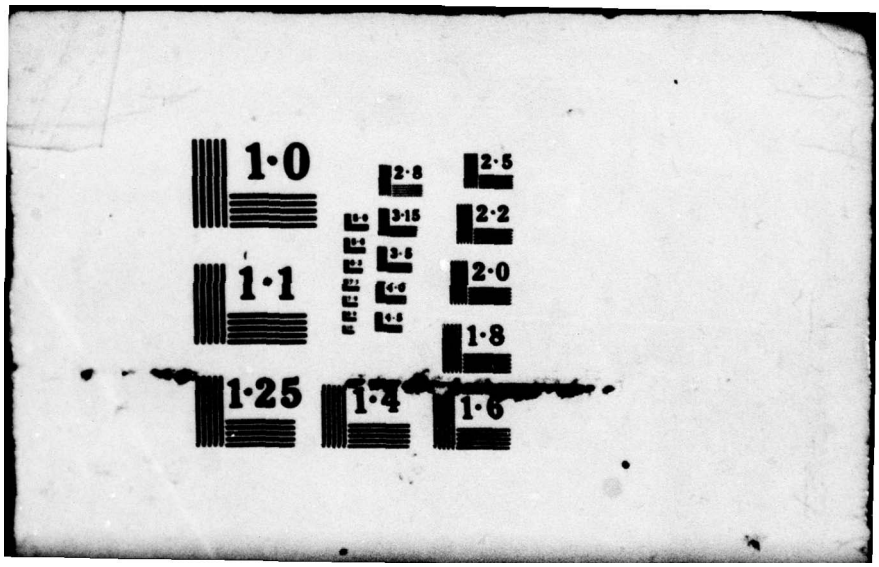
AD-A072 298 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
STUDY OF AUTOMATED COMMAND SUPPORT SYSTEMS.(U)
JUL 79 M C PEASE, D SAGALOWICZ N00014-77-C-0308
UNCLASSIFIED NL

1 OF 1
AD
A072298

END
DATE FILMED
9 79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5

STUDY OF AUTOMATED COMMAND SUPPORT SYSTEMS

Final Report

SRI Project 6289
Contract No. N00014-77-C-0308

July 9, 1979

By: Marshall C. Pease, Staff Scientist
Computer Science Laboratory

Daniel Sagalowicz, Computer Scientist
Artificial Intelligence Center

Computer Science and Technology Division

Prepared for:

Office of Naval Research
Department of the Navy
Arlington, Virginia 22217

Attention: Marvin Denicoff, Program Director
Contract Monitor
Information Systems Branch

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-1246

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

(6) Study of Automated Command Support Systems

5. TYPE OF REPORT & PERIOD COVERED

(9) Final Report, 1 Apr 78 - 31 Mar 79

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

(10) Marshall C. Pease
Daniel Sagalowicz

8. CONTRACT OR GRANT NUMBER(s)

(15) N00014-77-C-0308

9. PERFORMING ORGANIZATION NAME AND ADDRESS

SRI International
333 Ravenswood Avenue
Manlo Park, CA 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

(12) 56 p.

11. CONTROLLING OFFICE NAME AND ADDRESS

(11) Office of Naval Research
Arlington, VA 22217

12. REPORT DATE

July 1979

13. NO. OF PAGES

53

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Management, knowledge-base, modularity, planning, replanning coordination, organization, model process model, demons., and communications.

This study addresses ~~has sought~~ seeks

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The program has addressed techniques for the design of automated support systems for naval management. It ~~has sought~~ to apply advanced computer techniques developed by the artificial intelligence community to real-world command environments.

The managerial environment exhibits two major differences from most other application environments. First, the problems encountered by the manager are (over)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified 410 281

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

not routine or predictable. It is important that it be possible to tune the system to the actual needs as the problem develops. Second, the manager is the expert; he must be able to understand the system's behavior and the reasons for the actions it takes. In combination, these requirements indicate that the commander, or his delegate, ~~under his command~~, should be able to modify and extend the system even though they may have little knowledge of programming or system design. The crucial design issue, ~~therefore,~~ is to provide means by which the human user can exercise deep control without requiring a correspondingly deep knowledge of the system's implementation.

The primary device we have exploited to provide the user with effective control is to base the design on the explicit encoding of the system's knowledge as a set of models each of which contains the constraints and goals that describe necessary relations in the information that describes the situation in a specified component of the application environment. These models remain available to the user for his study and modification.

The implementation principles used to develop this design have been demonstrated in an experimental system called ACS.1 for automated command support. This system operates in the simulated environment of the command of a naval air squadron on a carrier. It assists the commander in the preparation of operational plans, the maintenance of the workability of these plans as events occur, the execution of approved plans, and the analysis of completed operations.

Previous reports have documented many of the techniques used in ACS.1. These reports are listed and their significance briefly discussed. Reports not previously mentioned are described in greater detail. Summaries are also given of results obtained in four major areas: the design and use of the message handler, the abstract concept of M-modules as a generalization of the main programming modules of ACS.1, active data bases, and the final design and behavior of ACS.1:

The message handler is a central switching facility that is also an important facility through which the user can exercise control over system operations at a very high level.

M-modules, developed as a generalization of the approach used in the planners and schedulers of ACS.1, appears to provide an important methodology for the design of hierarchical, knowledge-based systems. It is expected to be useful in many other application environments.

The concept of an active data base is concerned with using the system's knowledge to check not only the execution of approved plans, but also the validity of inputted data. Since no separate report on the active data base has been prepared, this area of the research is discussed in considerable detail.

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

ACS.1 has demonstrated the feasibility of a system based on the design principles and implementation methods that have been developed. It has shown that it is possible to provide automated support in the chosen area of management responsibility.

The final section discusses in considerable detail various research directions that, we believe, should be pursued. Future research on systems to provide support for management in many other areas and to meet other managerial needs can build on the results achieved here.

CONTENTS

I INTRODUCTION

This report is both the final report and the annual report for the Study of Automated Command Support Systems. The period covered is April 1, 1978 through March 30, 1979.

The research objective was to study ways of providing automated support to naval management. In particular, the intention has been to explore ways of applying advanced techniques developed in the artificial intelligence community to problems of management. We have concentrated on the managerial responsibility for creating operational plans, updating them as new information is received, monitoring the execution of the plans, and analysing the operations once the plans are completed. The particular context chosen for the major part of the research was the command of a naval air squadron operating from a carrier. A minor effort was devoted to task force movements, both in a transit operation and in the planning of a rendezvous with its supply vessels.

An important part of the research has been the development of the experimental system called ACS.1 (for Automated Command Support). This system has provided a vehicle in which particular problems of system design could be examined and approaches explored. It also served to demonstrate the resultant behavior.

A second important part of the research has been the development of the theory of what we call M-modules (for model-driven modules). M-modules are a generalization and abstraction of the program modules of ACS.1 called "planners" and "schedulers". This theory may have other important applications. It provides a methodology for the development of a highly modular program design that not only remains accessible to the user for his intervention in critical situations, but also allows him to modify or extend the system's scope of operations.

A third important research area was use of the system's knowledge for semantic checking of input data. It is recognized that the knowledge contained in the system for planning purposes can also be used for verification, determining that input data is reasonable and consistent with other data. There is little distinction between the use of semantic tests to check for input errors on the one hand, and tests to determine if an operational plan is being executed properly on the other hand. Data that do not fit a plan may fail either because the data are wrong or because the plan is not being executed properly. Since the validation of data in a large data base is a problem area long recognized as being of great importance, we have included work on semantic checking in our research.

II BACKGROUND

A. Contract History

The research program conducted under this contract continued and extended the program done under contract N00014-71-C-0210. That contract terminated on March 30, 1977 and a final report [1] was submitted in letter form dated June 20, 1977. The results obtained under that contract and the applicable technical reports and other relevant documents were summarized in that final report. An annual summary was also prepared and issued in January 1977 [2].

The work done under this contract from April 1, 1977 through March 30, 1978 was summarized in the technical proposal [3] for the continuation of the contract from April 1, 1978 through March 30, 1979. Publications issued during that period were as follows:

(1) Technical Report 13, "ACS.1: An Experimental Management Tool", issued early in 1977, summarized the concepts and implementation principles used in the design of the experimental system called ACS.1 (Automated Command Support).

(2) Technical Report 14, "The Schedulers of ACS.1", issued September 1977, documented in detail the design of the schedulers of ACS.1 and the concepts that have led to that design.

(3) Technical Report 15, "Planners of ACS.1", issued November 1977, documented the design of the planners of ACS.1 and the concepts that have determined that design.

(4) Technical Report 16, "The Message Handler of ACS.1", issued January 1979, documented the design of the message handler of ACS.1 and the concepts behind that design.

The experimental work performed during that period will not be discussed in detail here since it has been documented in those technical reports. The material presented here summarizes accomplishments since then.

B. Presentations

The following presentations were made during the last year:

(1) On January 18-19, 1978, D. Sagalowicz participated in an NRL Workshop on Semantic Error Detection, in Washington, D. C.

(2) On May 31 and June 1-2, 1978, D. Sagalowicz attended the International Conference on Management of Data in Austin, Texas. He participated in a panel discussion on natural-language access to data bases.

(3) On August 22-25, 1978, J. Goldberg and M. Pease attended the ONR Command and Control Workshop at the Naval Postgraduate School, Monterey, California.

(4) On December 6, 1978, J. Goldberg, M. Pease, and E. Sacerdoti attended an informal meeting of ONR contractors. Accomplishments of the various programs were reviewed and possible future directions discussed.

C. Papers and Reports

The following papers and reports were published during the past year:

(1) D. Sagalowicz, "The Message Handler of ACS.1," Technical Report 16, SRI International, Menlo Park, California, 94025, January 1979. This report documents the design of the message handler, and together with other technical reports [4-5] describing other parts of the system, completes documentation of the current system.

(2) M. Pease, "ACS.1: An Experimental Automated Support System," *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-8, No 10, pp 725-735, October 1978. This paper describes the system as a whole and discusses its design principles and some of its important implementation features.

(3) M. Pease, "M-Modules: A Design Methodology," Technical Report 17, SRI International, Menlo Park, California, 94025, March 1979. This report develops the theory of M-modules as a general methodology of programming complex systems for a range of possible applications. The concept of M-modules is derived as a generalization of the design of the planners and schedulers of ACS.1.

In addition, D. Sagalowicz is preparing a paper for submission to an as yet unselected journal on artificial intelligence. This paper

will discuss the design of the ACS.1 system and its underlying concepts from the artificial intelligence perspective.

III TECHNICAL RESULTS

Work during the past year has advanced the research in three main areas. These are discussed in the following subsections. Presented, also, is a discussion of the experimental system, ACS.1.

A. Message Handler

The message handler is a central switching module through which pass all communications between the planners, schedulers, the data system, and the user. Because of its central location, we recognized its potential for user control. To realize this potential, the message handler should be able to modify messages passing through it. For example, it might be directed to change the destination of a message. Whenever a message is received from module A (say, a planner) addressed to module B (perhaps a scheduler) asking for some action (such as the assignment of a resource to a plan), the message handler might be directed to send the message to a different module or to the terminal. In the latter case, the user can determine what action to take (what resource to assign to the plan) and enter his decision. His answer is put into proper form and returned to module A as if it had been returned by module B. Module A does not even need to know that module B has been cut out of the loop.

The ability of the message handler to modify and redirect messages is an important aspect of system flexibility, allowing it to be tuned to the manager's immediate requirement. In the example mentioned previously, the user is intervening, taking over the functions normally handled by module B. He may do this because of exceptional circumstances, knowing that the rules used by module B are not applicable under the existing conditions, or he may do so because of particular requirements not represented in the knowledge used by module

B. For example, he might have his own reasons for insisting that a particular pilot should fly some mission. Whatever the reason, the capability gives the manager an easy way to intervene and to take over some part of the system's responsibilities.

In another kind of situation, the message handler can switch between two modules that do similar tasks but use different knowledge or rules. For example, different rules might be needed during training than under combat conditions. Two pilot schedulers could be created: one for use during training, the other under combat conditions. To switch between them, the manager need only enter a command to the message handler.

In still a different situation, the facility can be used to debug and verify a new module before relying upon it. All messages to and from the new module can be routed to the terminal and held there until approved by a human expert. This feature is particularly important for the incremental growth of a system. It allows new responsibilities to be assigned to the system and checked out in detail under operational conditions. If errors remain in the encoding of the new knowledge, they can be caught without allowing a disastrous situation to develop.

The message handler has many points of structural similarity to the planners and schedulers. The theory of M-modules, discussed in the next section, has been developed partly in recognition of the existance of such structural similarities among entities (planners, schedulers, the message handler, and even the data base management system) that serve very different functions.

To illustrate the analogies, we can recognize that the message handler is an autonomous program module driven by the set of commands that have been given it. The list of commands can be regarded as a model of what the user wants the system to do, and so is serving an analogous function to the process model in a planner or to the resource model of a scheduler. Also, the messages queued in the message handler at any time can be identified as a set of values that instantiate the commands. The queue of messages, therefore, is analogous to the set of

plans in a planner, or to the scroll table of a scheduler. Finally, the message handler has the set of functions required for its operations, as do the planners and schedulers. These functions use the knowledge encoded in the commands to generate their actions. In summary, the message handler contains elements that are analogous to those that define a planner or scheduler. It is, in fact, an M-module as the term is defined in the next section.

The design details of the message handler have been given in Technical Report 16.

B. M-Modules

An M-module, or model-driven module, is an abstract program module that was developed as a generalization of the planners and schedulers of ACS.1. The design principles embodied in ACS.1 and the implementation techniques used there could have broad application. Examples of other application environments in which these principles and techniques appear to be useful include $\frac{3}{C}$ (command, control and communications), war gaming at the theater level, and the modeling of complex logistic requirements. An abstract description of ACS.1 was necessary so the applicability of the design principles to such diverse environments could be examined. The concept of an M-module as an abstract description of the planners and schedulers of ACS.1 has been developed to meet this need.

An M-module is a real or virtual program module whose behavior is controlled by knowledge that it contains in the form of a model. This model holds the knowledge used by the M-module in declarative form, not embedded in procedures. The procedures used by the M-module are unspecialized; when called, they must examine the model to determine what their exact behavior should be. They translate the declaratively encoded knowledge into the operations needed to express the implications of that knowledge.

The retention of the knowledge in explicit declarative form as a model helps keep it accessible and intelligible to the manager who is not expert in programming. It is an important feature in enabling him to control system behavior according to his needs, and to tune the system as those needs change.

The M-module also retains all the values necessary for the full exercise of its responsibilities as defined by its knowledge. The set of values is accessible only from inside the M-module, although the M-module will accept and respond to requests asking for particular values. This restriction on access is necessary to ensure that the set of values remains internally consistent according to the restrictions and constraints expressed or implied in the model. For example, the resource model of a scheduler defines what is an assignment conflict. It is the responsibility of the scheduler to ensure that a conflict does not arise, or to initiate appropriate corrective measures should one do so. This requirement means that the scheduler must contain all the available information about the future uses of the resources it handles. It also means that no other process elsewhere in the system can be permitted to enter or modify the scheduler's information except by invoking the scheduler's own processes, which can also enforce the rules that prevent conflict.

A useful feature of the M-module approach is that it permits the system to be developed as a hierarchical structure. It is possible to collect M-modules together with some additional components to make a single M-module at what is called a higher level. Indeed, the entire ACS.1 system can be seen as a single M-module. Conversely, it is often possible and useful to regard a given M-module as being built from a number of constituent M-modules and some additional components. A planner, for example, can be regarded as composed of subplanners, each of which is responsible for a particular task or assignment in the operation handled by the planner. These subplanners are regarded as M-modules at the next lower level.

The hierarchical decomposition so obtained is a structural one that should not be confused with the conceptual decomposition according to level of detail among M-modules at a given level. For example, the mission planner of ACS.1 understands a mission as being composed of five tasks (A/C-PREP, PILOT-BRIEF, FLIGHT, A/C-SERV, and PILOT-DEBRIEF) and two assignments (PILOT and A/C, or aircraft). A plan for a mission will include values for the start and end times of each task and assignment and the name of each resource assigned.

The mission planner is composed of a set of subplanners, one for each task and assignment, and some additional components. Each subplanner uses the appropriate part of the mission planner's knowledge and is responsible for its particular values. For example, the subplanner for A/C-PREP is responsible for maintaining the start and end times of this task in each plan. However, although the A/C-PREP subplanner may have an estimate of how long this task is likely to take, it does not have the detail that would allow it to plan the task. It does not know that preparation of the aircraft requires that it be armed, fueled, and inspected and that these operations require the assignment of maintenance personnel. The detailed planning of the aircraft preparation must be done by another planner with a process model describing the task in this additional detail. The A/C-PREP-planner knows how to plan the task in detail; the principal function of the A/C-PREP subplanner of the MISSION planner is to know that the task must be planned and to initiate the process. It is, in other words, the subplanner that must cause a message to be sent asking the A/C-PREP-planner to create a plan.

There are, then, two ways of decomposing the system's knowledge--one by level of detail among the various planners, the other by the scope of attention in the subplanners. In ACS.1, both decompositions are used simultaneously, and both contribute to the system's value. The decomposition into levels of detail is of particular importance as it exploits various attributes of the application problem in order to make the system behavior more intelligible to the manager or his delegates.

The hierarchical decomposition into levels is important because it allows us to impose a discipline on the interactions between M-modules. We are able to ensure that the system will respond to any input in an orderly and systematic way. We have discovered, for example, that it is possible and natural to impose conditions on inter-M-module communications to ensure that the system stays free of deadlock. The same conditions also facilitate the design of the system to prevent instability.

The design of systems based on M-modules combines the concept of knowledge-based systems developed by the artificial intelligence community with some of the structured programming methods of computer science. The details of the concept, along with their implications for system design, were discussed in Technical Report 17. The concept is seen as leading to a design methodology that may have considerable significance in diverse application environments.

C. Active Database

Our research has addressed the issue of providing semantic descriptions of databases. It has used process models similar to those used in the planners of ACS.1. It has sought to develop means of providing semantic checks to determine the validity of data, and to verify that current plans are being executed satisfactorily.

Even though the idea of using semantic knowledge to improve the checking of database assertions has been mentioned extensively in various conferences, actual published results in this area are very limited. The most significant papers are [8,9] which formally introduce the notion of semantic error detection. In the latter paper [9], the authors also attempt to improve on the efficiency of such semantic monitoring by trying to decide whether some of the checks may be by-passed. In our approach, we attempt to give the user full control on what semantic checks need to be performed, by the use of descriptive "semantic update models."

A typical system scenario starts with the user requiring the system to prepare a plan for some operation. Desired objectives are given the system as part of this requirement--e.g., prepare a plan for a mission to strike a specified target at a specified time and to leave the target area at a specified later time. A plan is developed employing the full capabilities of the planners and schedulers, as well as all available information about other expected uses of the needed resources. This plan is returned to the requester for his approval. Once approved, the plan is transmitted to the database management system (DBMS) which accepts responsibility for monitoring its execution. The database continues to receive inputs giving updated information about operations. The DBMS checks these inputs, first to determine that they are not obviously invalid, and then to determine their implications for the plan. Should it be determined that execution is diverging significantly from the plan, the DBMS must recognize the fact and decide what to do about it. It will, in general, alert the commander to the situation. If it has been given the authority, it may initiate replanning by requiring that the appropriate planner resume responsibility.

There are several ways for the DBMS to monitor a plan or input data. The simplest verification checks are syntactic, verifying the data format. Direct comparison checks against expected values are almost as simple, unless complex rules about what errors can be tolerated are needed. For example, intermediate check points may have been identified in the plan for a task force transit operation. If the current position of the task force is maintained with its accuracy being reestablished periodically, it is simple to verify that the task force has reached a checkpoint at a time not later than that designated in the plan.

Complications can arise even in direct comparison checking, however. One important and potentially difficult problem is to identify the checkpoints that should be used. A system such as ACS.1 includes a process model for the transit operation. It seems reasonable to expect the system to use this model to generate its own checkpoints. The

transit operation is described in the process model as a sequence of "legs," each of which specifies a fixed course and speed. It is reasonable to use the start or end time and location for each leg as a checkpoint. However, if a leg is very long, we may need to use intermediate points as well. Also, during transit of a complicated body of water, there could be many short legs specified. In this case, we might want to use only the point of entry into this area and the point of departure, ignoring the component legs. Automatic means for determining the checkpoints that will be used can require the identification and use of some quite unobvious rules.

Complications also arise because the planning model may not conform to the way in which data are gathered. In effect, the execution model may be quite different from the observational model. The transit of a task force is naturally modeled as a sequence of legs, each perhaps specified by course and speed from an initial time. To this can be added the position at the start of each leg. However, this addition introduces a degree of overspecification. Suppose the actual transit operation varies somewhat from the plan, with no leg being executed exactly as planned. Should the check at the start of a leg be made against the originally computed start position for that leg? Should the start position be recalculated from the start of the transit using the actual legs and computing to the planned start time of the leg? These two procedures will lead to different checkpoints. The former seems the most reasonable one to use, but how can we ensure that it will actually be the one used?

Furthermore, even if we do agree on a sequence of checkpoints, they may not correspond to the way in which data are gathered. For example, the location of the task force may be determined and reported only at times that have no direct relation to the start or end time of a leg. We would be very reluctant to prohibit the entry and use of such data. We would like the system to accept and use any data that are logically sufficient.

Finally, there is the problem of efficiency. The planners and schedulers of ACS.1 can afford to sacrifice efficiency, at least within limits, to do whatever is desirable. This is much less true in a database where the volume of information to be stored does not permit an inefficient use of memory resources. The volume of transactions involving the entry or updating of information is likely to make it very undesirable to depend upon complicated computational processes.

To resolve these problems, we use a sequence of models of increasing detail and difficulty. The initial checks are done using the simpler models. Only after an update has been found acceptable according to the simpler models, and only if the system is not too heavily loaded, are the more complex and detailed models used. This use of hierarchies of knowledge is closely related to the idea of a hierarchy of abstraction spaces developed by Sacerdoti [7].

To illustrate, suppose data are entered that purport to be an updated report on the task force's position. The first question after performing syntactic tests might be whether the position is possible, given the task force's previous position and its possible speed. This is a simple check; it uses only the times and distances. If the data pass this check, the next question might be whether it is possible, considering the neighboring land masses, reefs, or other obstacles. Only if the data pass these tests is it worthwhile to consider if the data are consistent with the original plan.

In many cases, it is possible and convenient to use a pair of models, one causing the data to be accepted without further checking, the other causing its immediate rejection. In this case, further checking is done only if neither test gives a conclusive answer. To illustrate, position data can be accepted without further test if it is sufficiently close to the planned location; it can be rejected out of hand if the distance from the last reported position is too great (at least providing it is judged that the last position report was accurate). Only if neither test is conclusive do we need to consider that the data may be accurate but the plan is not being followed.

Each model in the test sequence has actions associated with the various possible outcomes. The action may be to execute the next test (or pair or set of tests) in the sequence. It may be to challenge or reject the data, to accept the data without further checking, or to seek help from the responsible human. If further testing is indicated, the action will also specify whether the new tests are done immediately or on a when-convenient basis. In the latter case, the action specifies how the data are to be handled in the meantime. For example, the data might be given a warning to indicate the conditionality of its acceptance.

In Appendix A, we explain how our current experimental system checks updates using update models set up by a user. In Appendix B, we give an example of such an update model in a pseudo-English format. In Appendix C the actual update model is listed. Three examples of update checking are presented in Appendix D.

It can be observed that the execution monitoring could be implemented using M-modules. A given plan could be attached to an M-module. This M-module would monitor the execution of plans of the same type, where the type is an equivalence class based on the verification tests that are used. The models that specify the various tests would represent the knowledge encoded in the module. The procedures of the module would be those that actually apply the tests and that initiate the actions resulting from the outcome of the test. As indicated in Appendices A through D, our actual implementation did not use M-modules.

To summarize, an active DBMS is possible using a graded sequence of models for testing input data, both for its semantic consistency and for its implications about the execution of plans contained in the DBMS. We have indicated a sequence of models that could be applied to the transit of a task force. Similar decompositions of the applicable knowledge could be made to operations that involve, for examle, the rendezvous of supply vessels with a transiting task force. The result would be a sequence of models each of which specifies a test and one or more actions. The actions indicate what is to be done depending on the outcome of the test with which it is associated.

D. The Experimental System

The experimental system, ACS.1, developed under this contract, has shown the system concept and its associated techniques to be feasible. It has suggested the value of the design concept for systems intended to support management in the planning and executing operations. It has demonstrated that the design concept can be implemented. As developed, the system is knowledge-based, model-driven, and exhibits considerable inferential capability while remaining controllable by the manager in a flexible and adaptable way.

Flexibility was considered to be of prime importance to keep the system responsive to the manager's needs. It must be assumed that the manager's needs will change as the operational situation develops, as the organization and its policies evolve, and as managerial goals change. The system must be flexible and adaptable to allow its being tuned as needed in response to these changes. The following features of ACS.1 use techniques that directly address the requirement of flexibility and adaptability:

- The system, as seen by the user, is highly modular with each module handling a specific area of responsibility.
- Coordination among the modules is maintained through a process of negotiation in which each module retains control over its own area of responsibility. A plan can be regarded as a set of coordinated commitments by the various modules with each module retaining a continuing responsibility for its own commitments.
- Human organizations provide useful prototypes for system design; they can suggest a natural and useful decomposition of the system into modules, and can suggest convenient boundaries for the modules' individual areas of responsibility.
- The message handler provides a central node at which the user can intervene to monitor or switch particular system functions. It is a convenient module for the execution of managerial decisions that affect the major operations of the system.
- The functions of ACS.1--planning, maintenance of approved plans, monitoring their execution, and supporting the analysis of completed operations--are well differentiated. ACS.1 demonstrates the feasibility of providing separate means for the support of these functions.

- Since both the process and resource models are declarative expressions of the knowledge used by the system, this knowledge remains accessible in understandable form for modification, adaptation, and evolution under the direction of the manager.
- Data structures have been developed that match the requirements of the modules that use them. The particular A-lists (association lists) that encode the models and the scroll tables of the schedulers are examples.
- The procedural forms that enforce the system's knowledge are implemented by what are called demons. (Demons are processes invoked by changes in data, or variable assignments, that satisfy a predicate specified by the demon.) It was found feasible to write functions that automatically generate these forms as the need for them is implied by the models. This has proved to be a very powerful technique that is applicable to a wide variety of situations.
- Procedural forms, or demons, have also been found useful for other purposes such as monitoring for underloading or overloading of particular types of resources during specified periods, maintaining running totals, and controlling actions that should be initiated when an accumulated total passes some value. Again, demons have proved to be a powerful and versatile tool.

ACS.1 has demonstrated the possibility of obtaining the qualities needed in a system to support managerial planning. In particular, it has demonstrated the possibility of a system over which the manager can exercise a high degree of control with minimal attention to its implementation details. It is a flexible, knowledge-based facility that can support managerial needs in many areas of command.

IV RECOMMENDATIONS

The immediate direction of further research should be toward providing automated support for cooperative problem-solving in managerial environments. By cooperative problem-solving, we mean the joint effort of several participants--they can be individuals, organizations, or computers--to solve a problem in a way that depends on each participant having or acquiring some understanding of the goals, constraints, and viewpoints of the others. It requires the sharing of knowledge about the problem and about each other, and the use of this shared knowledge in the search for a mutually acceptable solution.

We recognize the growing need for improved ways to obtain and support cooperative-problem solving in management and command. The need for cooperation is an ancient one, as is the difficulty of obtaining it under crisis conditions. However, the need is becoming ever more important with the increasing complexity of the operations being managed, the possible global impact of operations, the speed with which crises can arise or changes occur, the geographic disper-sion of resources and other components of a problem or a solution, and the diversity of activities that can be involved. Taken together with the experience that has been gained with ACS.1 and the general and growing ability of the artificial intelligence community to handle complex real-world problems, it is appropriate and timely to address the research issues involved in providing support to cooperative problem-solving in the management environment.

The advances made on this program, when taken together with the progress of the artificial intelligence community in the representation and use of knowledge, indicate the possibility for substantial progress in understanding and automating cooperative problem-solving. We recommend that the current program on the use of advanced computer

techniques in management be expanded to include research towards the goal of understanding how to provide automated means that can support cooperative problem-solving.

The program has lead to a number of fundamental concepts and techniques that have substantial promise in other areas. The concept of M-modules and their use as a general programming methodology is an important example. The methodology leads to hierarchical systems of modules with a number of interesting properties. The immediate purpose for this program was to allow a user who may not be an expert programmer to control the behavior of the system. It does this by retaining knowledge in the form of a model that remains accessible for modification and adaptation. In addition, the technique permits the use of general functions whose detailed behavior is determined by the model. Hence the functions can be made common to the system and used by whatever M-modules need them. This is important in making it easy to construct new modules or to reconstruct existing modules that may have been destroyed by a fault. In particular, if the system is distributed among a number of processing centers, either in a multiprocessor facility or in a geographically distributed network, the assignment of modules to processors can be changed easily. The effect is to move a module from one processor to another; the actual mechanism is to recreate the module in the new location.

The M-module methodology has another important implication for multiprocess operations: it imposes a tight discipline on intermodule communications that allows the system to operate asynchronously. In particular, it restricts communications to pairs of modules that are siblings, i.e., M-modules that are immediate subordinates of a single M-module. Further, it imposes conditions under which communications can occur between siblings--specifically that they and all their siblings must be in exterior states. (An exterior state of an M-module is one in which its set of values is fully consistent with the rules and constraints of its model.) The effect of this condition is to ensure that deadlock cannot occur. The condition also seems to facilitate the

avoidance of starvation, although we have not been able to find conditions that guarantee the avoidance of starvation.*

In summary, the M-module methodology is an important contribution that merits further study and development. It can be useful for the design of knowledge-based systems to support users who are expert in the applicable field. It is also of great potential importance to multiprocessor systems, allowing them not only to attain high reliability and survivability, but also to be easily expandable and adaptble to meet changing requirements and conditions. Further studies to explore the latter possibilities should be pursued.

Among other accomplishments of the program, the role of the message handler in providing the user with an additional measure of control is noteworthy. The implementation details of the message handler are of some interest. Of greater interest, however, is the system concept that it embodies. It can be regarded as a means through which the user can introduce meta-commands, directing the system as to how it should command its own actions, or respond to commands within or external to itself. (The message handler can be directed to modify messages to or from the user himself.) It introduces a new level of user control. The possibilities for the use of this level of control should be examined in more detail.

Finally, various implementation mechanisms developed for the experimental system may have wider application. The technical reports produced under the contract and its predecessor have described these mechanisms. Particular note should be given to the process models used

---

* Deadlock is the condition when no process can complete its task because each incompleted task needs the results of some other deadlocked task. Process A may need the results of process B before it can complete its assignment, and process B need the results of process A. Neither can continue. Starvation is a weaker condition in which there may be no logical reason why all processes cannot complete their tasks, yet there remains a finite probability that some will not be completed in any finite time. In this context, module A may need results from module B. While there is nothing that blocks module B from furnishing the needed information, module B may not do so for an indefinite period of time.

by the planners, the resource models used by the schedulers, the forms used to record the information retained by an M-module (e.g., the scroll tables used by the schedulers), and the demons used as the procedural form of the constraints encoded in declarative form in the models.

It would be of considerable interest to evaluate these advances for possible use in a multiprocessor environment that may be distributed as well. For example, the message handler may have an important role to play in a distributed system in providing a central, coordinating entity with which the user can interact. However, the development of a distributed message handling capability that could fill similar functions without being vulnerable to the failure of a central hardware unit would have advantages.

The different kinds of models that should, or could, be used to control a multiprocessor system to serve other application needs in a distributed environment should be studied. The roles of the models in an M-module are sufficiently identified to provide a firm framework for such a study.

The effects of the forms used to record and present the information used by an M-module should be evaluated for other applications, as well as for the particular requirements of multi-processor and distributed environments. The M-module methodology requires that a given datum or variable assignment be retained by only a single M-module. Hence, the methodology avoids the problem of maintaining consistent updates over a distributed system. If it is duplicated in some more general data system, maintenance of the duplicate information is a separate issue, not a part of the M-module structure.

Finally, the role of demons in maintaining the consistency of an M-module's set of values versus the rules and constraints of its model needs to be expanded to take into account the needs of a multiprocessor and distributed system. In particular, demons can be used to generate and control the messages that will provide the necessary coordination.

In conclusion, the work done under this contract has led to the development of a number of new concepts and techniques that have great potential. They are applicable to a number of application environments besides the planning and scheduling one addressed by ACS.1. Perhaps of greatest interest as a direct expansion and continuation of past work, they appear to lay the groundwork for research into systems for the support of cooperative problem-solving. In a different direction, these concepts and techniques appear to have great potential for multiprocessor and distributed systems. We recommend that these possibilities be vigorously pursued in future research programs.

REFERENCES

1. Letter to Mr. Marvin Denicoff, Office of Naval Research, from Jack Goldberg, Director, Computer Science Laboratory, SRI International, dated June 20, 1977.

2. M. C. Pease, J. Goldberg, and D. Sagalowicz, "Annual Summary, Research on Large File Management Information Systems" Stanford Research Institute, Menlo Park, California (January 1977).

3. M. C. Pease and D. Sagalowicz, "Study of Automated Command Support Systems," Proposal for Research, Part One--Technical Proposal, SRI International, Menlo Park, California (6 March 1978).

4. M. C. Pease, "The Schedulers of ACS.1," Technical Report 14, SRI International, Menlo Park, California (September 1977).

5. M. C. Pease, "Planners of ACS.1," Technical Report 15, SRI International, Menlo Park, California (November 1977).

6. D. Sagalowicz, "The Message Handler of ACS.1," Technical Report 16, SRI International, Menlo Park, California (January 1979).

7. E. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, Vol. 5 No. 2, pp. 115-135, Summer 1974.

8. M. Hammer, "Error Detection in Data Base Systems," NCC, pp. 795-801, 1976.

9. M. Hammer and S. K. Sarin, "Efficient Monitoring of Database Assertions," to appear in ACM Trans. on Database Systems.

Appendix A

USE OF UPDATE MODELS

Appendix A

USE OF UPDATE MODELS

This appendix describes the current implementation of our update checking system. This implementation is based on the use of update check models which are given by the user. The system simply attempts to follow the directions given in those models. The purpose of this idea is to give the user an easy tool to implement complex hierarchical checks, as explained previously.

The models use two sources of data--a time sequence model and the database information about the tasks.

The time sequence model is an ordering, by time, of the tasks that will be checked. For example:

A MISSION can be broken into the sequential tasks, PRE-FLIGHT, FLIGHT and POST-FLIGHT. MISSION is the SUPERTASK of PRE-FLIGHT, FLIGHT and POST-FLIGHT and the latter are MISSION's subtasks. PRE-FLIGHT, the task immediately preceding FLIGHT, is called FLIGHT's ANTECEDENT task.

PRE-FLIGHT consists of AIR-CRAFT-PREPARATION and PILOT-BRIEFING, two tasks that can be done concurrently. PILOT-BRIEFING is AIR-CRAFT-PREPARATION's CONCURRENT task.

The remaining tasks in MISSION are broken down in the same manner. The terms SUPERTASK, ANTECEDENT, and CONCURRENT are used in the model of Appendix B.

The database of time information about these tasks includes the start time, end time, minimum duration, maximum duration, and any other relevant information.

An update check model is a list of the form:*

---

* The "/" sign indicates an option between several alternatives.

```
[modelname modelarg/(modelarg1 modelarg2 ...)
        (test1 (result1 action1 action2 ...)
               (result2 action1 ...) ...)
                .
                .
                .
        (testn (result1 action1 ...) (result2 action2 ...) ...)]
```

where: modelname is the name of the model, modelargs are names of the arguments used in the model, and testi are tests to be performed.

A test is of the form: (arg1 fn arg2/(arg2 arg3 ...)) where: arg1 is an item we are getting information for, or comparing to some value; fn is either an information acquisition function (such as DBFETCH, which fetches a value of a field from the database or TSMFETCH, which fetches the value of an attribute in the time sequence of the model of the task under consideration) or a comparison function; arg2/(arg2 arg3 ...) is the information to be acquired or the limits to be met in a comparison.

For example, in Appendix C., we use the following test: (CHECKTIME *INRANGE ((*IDIFF START 30)(*IPLUS START 30))) to check that the actual start time is close to the planned start time of the task.

The tests specified in the update check model are made in sequence and the result checked against the permissible results. If the result matches a permissible result, the specified action(s), actioni, will be taken. If there is no match or there are no actions, the next test in sequence is made. Falling out of the model is the same as ending the check.

An action can be:

- another model, ie, another series of tests will be performed.
- success
- failure
- queue and continue processing. Part of a model will be "queued", typically, because data are missing and must be waited for. The rest of the model will be tested when the data are entered.
- stop check now

It is possible to implement various "levels of checking". The idea is to test only at a specified level of checking and do more complicated tests during background times, i.e., when the system is lightly loaded. This can be implemented either by specifying different models for different levels of testing or by selecting the appropriate portion of a model for the current level using a

```
(SELECTQ CHECKLEVEL (1 ...) (2 ...) ... NIL)
```

in the model. If we fail at a low level of checking, we may queue the failure until the program has time to do a more thorough check or ask the user if he wants a more thorough check. We can specify that CHECKLEVEL be tied to whether the user can be fully trusted or not. If the user is a production program, we would have high confidence that its information is adequate, and will reduce the amount of checking to be done on-line. On the other hand, if the user is an interactive user, it appears more sensible to increase the amount of checking before accepting any update.

As indicated above, in many cases, a test has to be delayed: this may happen for example when data needed to do the test are missing or when the check level is too low for the current level of activity. In those cases, the test is delayed and put in a queue. The next paragraphs present the treatment of this queue.

The function QUEUE will put models waiting for information from the database on the queue in the form:

```
(TIMELIMIT  ((VAR1 . VAL1) (VAR2 . VAL2) ...)  MODEL)
```

where TIMELIMIT is the number of millisecondss to leave the entry on the queue, ((VAR1 . VAL1) (VAR2 . VAL2) ...) are the data values needed for MODEL, MODEL is the name of the model or part of the model waiting to be executed.

The function EXMODEL will execute the major model (for example, CHECKSTART) and put any deferred checks on the queue. After completing the model, EXMODEL will call PROCESSQUE to process the entries on the queue until the queue is empty.

In order to process the queue automatically, we have considered putting demons on missing data. When data come in, the demon would wake up and process the entry. A priority code could be placed on the entry or the ordering of the queue could be the priority. These ideas have not been actually implemented.

During the processing of the model, whether it had been previously queued or not, the processing functions automatically throw away models or parts of models that have already been checked before queuing or requeing the models. This avoids having to reprocess tests that have already been made.

Appendix B

ENGLISH VERSION OF THE UPDATE MODELS

Appendix B

ENGLISH VERSION OF THE UPDATE MODELS

As discussed in Appendix A, our update check algorithms use an update check model given by the user. The general format we will use to describe a model is as follows:

```
MODELNAME  MODELARGS
  test1
  test2
    .
    .
    .
```

Descriptive comments will be introduced between the tests, indented to make it clear that these are comments, not part of the formal description.

CHECKSTART is described below to illustrate an update model. The purpose of CHECKSTART is to check any update to the starting time of a task.

CHECKSTART NIL

CHECKSTART does some gross checks on the data in hopes of terminating the update check without going into a detailed check. It first fetches from the database the expected values for the planned schedule of the task:

fetch START, END, MIN.DUR, MAX.DUR of CHECKTASK from database.

NIL: Print message indicating any missing data and continue in sequence.

It then compares the update value with the planned value:

is CHECKTIME within + or - 30 of original start time?

NIL: Print msg indicating check failure and terminate. NODATA: Print msg, put CHECKSTART on the queue and continue in sequence.

It next checks that we still have time to execute the task:

is new duration within MIN.DUR and MAX.DUR?

NIL: Print msg indicating check failure and terminate. NODATA: Print msg, put CHECKSTART on the queue if not already there and process queue (wait for data from database).

Finally, it checks if the task is started later than originally planned: if not, we must then check SUPERTASK, i.e., the task that includes the current task as a subtask:

is CHECKTIME greater or equal to the original start time?

T: Print OK and terminate. NIL: (EXECUTEMODEL 'CS.SUPERTASK CHECKTASK)

Check SUPERTASK of CHECKTASK.
NODATA: process the delayed queue

The last test uses CS.SUPERTASK. This model is as follows:

CS.SUPERTASK TASK

CS.SUPERTASK determines whether the update is within the start time of SUPERTASK of TASK. If it is, and there are no ANTECEDENT tasks for TASK, the update can be made with no conflict; otherwise, a check is made on the start times of CONCURRENT tasks of TASK to see if the update occurs after the start of a concurrent task. If the update is earlier than the start time of SUPERTASK, the end times of the ANTECEDENT tasks of SUPERTASK are checked to see if any preceding task ends after the update.

It first gets the SUPERTASK from the time sequence model:

get SUPERTASK of TASK from time sequence model

NIL: Print msg indicating check failure and terminate.

It checks that we know the start time of the supertask:

get START time of SUPERTASK from database

NIL: Print msg, put current model (beginning with this test) on the queue and process the queue.

It then compares the current update with this start time: if the update time is earlier, then we test the task that immediately precedes the supertask:

is CHECKTIME after START of SUPERTASK?

NIL: (EXECUTEMODEL 'ANTECEDENT (LIST SUPERTASK SUPERTASK))

If, on the other hand, the value in the update is after the start time of supertask, the task that immediately

precedes it is checked. If there is none, the update is checked OK. Otherwise, it tests any tasks that are executed in parallel with the current task:

get antecedent of TASK from time sequence model

NIL: Print OK and terminate.

T: (EXECUTEMODEL 'CS.CONCURRENT (LIST TASK SUPERTASK))

Check TASK's concurrent tasks.

This uses the model CS.CONCURRENT, given as follows:

CS.CONCURRENT (TASK SUPERTASK)

This model tests an update with respect to the tasks that can be executed in parallel with the current task. If there is any CONCURRENT task whose start time is earlier than the the update, the update can be put in the database without conflict. If there are none, the end times of the ANTECEDENT tasks of TASK are checked to see if any preceding task ends after the update.

It first gets the concurrent tasks from the time sequence model. If none, the antecedent task is checked for TASK:

get CONCURRENT tasks for TASK from time seqeuence model

NIL: (EXECUTEMODEL 'CS.ANTECEDENT (LIST TASK SUPERTASK))

If there are concurrent tasks, they must be checked to see if any started before the start time being updated. If yes, update is acceptable. If none, the task immediately preceding the task being tested is checked:

does any CONCURRENT task start before CHECKTIME?

T: Print OK and terminate.

NIL: (EXECUTEMODEL 'CS.ANTECEDENT (LIST TASK SUPERTASK))

Check antecedent of TASK.

NODATA: Print msg, put current model(beginning with this test) on the delayed queue and process that queue.

The model, CS.ANTECEDENT, used in the last test is as follows:

CS.ANTECEDENT (TASK NEXTTASK)

This model is used to check the end time of the task that immediately precedes the current task. If the preceding task ends before the update, the update can be put into the database with no conflict; if it ends after the update, there is a conflict. If there are no antecedent tasks for TASK, the SUPERTASK of NEXTTASK is checked using model CS.SUPERTASK:

is TASK = NIL?

T: Print msg indicating check failure and terminate.

It first finds what task precedes the current task. If none, the SUPERTASK of NEXTTASK is checked using model CS.SUPERTASK:

get ANTECEDENT for TASK from time sequence model.

NIL: (EXECUTEMODEL 'CS.SUPERTASK NEXTTASK)

If there are antecedent tasks, it checks their ending times which ought to be before the start time being tested:

does any ANTECEDENT task end after CHECKTIME?

T: Print FAIL and terminate.

NIL: Print OK and terminate.

NODATA: Print msg, put current model(beginning with this test) on the delayed queue and process that queue.

This completes the presentation of the update checking model we have used during our tests. The next appendix gives the actual listing of this model.

Appendix C

LISTING OF THE UPDATE MODEL

Appendix C

LISTING OF THE UPDATE MODEL

The actual functions used in the update models described in Appendix B are as follows:

```
[CHECKSTART NIL (CHECKTASK DBFETCH (QUOTE (START END MIN.DUR
                                           MAX.DUR))
                          (NIL (PRIN1
                "Missing data in START, END, MIN.DUR or MAX.DUR.")
                               (TERPRI)))
   (CHECKTIME *INRANGE ((*IDIFF START 30)
                        (*IPLUS START 30))
                       (NIL (ALERT
                    "New start time not close to old start time."))
                       (NODATA (QUEUE "Missing START" NIL NIL
                                      CHECKSTART NIL)))
            ((*IDIFF END CHECKTIME)
             *INRANGE
             (MIN.DUR MAX.DUR)
             (NIL (ALERT
          "New duration not within duration limits of TASK."))
             (NODATA (AND (NEQ START (QUOTE NODATA))
                          (QUEUE "Missing END, MIN.DUR or MAX.DUR"
                                 NIL NIL CHECKSTART NIL))
                     (PROCESSQUE)))
    (CHECKTIME *IGEQ START (T (ALERT "OK!"))
               (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                                  CHECKTASK))
               (NODATA (PROCESSQUE]

[CS.SUPERTASK TASK (TASK EQ NIL (T (ALERT
                    "FAIL.  TASK is NIL.  Can't go any further.")))
    (TASK TSMFETCH (QUOTE SUPERTASK)
          (NIL (ALERT "No SUPERTASK for TASK.")))
    [SUPERTASK DBFETCH (QUOTE START)
          (NIL (QUEUE "No start time for SUPER.TASK"
                      (TASK SUPERTASK)
                      (PROCESSQUE]
    [CHECKTIME *IGEQ START (NIL (EXECUTEMODEL
                                 (QUOTE CS.ANTECEDENT)
                                 (LIST SUPERTASK
                                       SUPERTASK]
                  (TASK TSMFETCH (QUOTE ANTECEDENT)
                        (NIL (ALERT "OK"))
```

```
                    (T (EXECUTEMODEL (QUOTE CS.CONCURRENT)
                                     (LIST TASK SUPERTASK]

[CS.CONCURRENT (TASK SUPERTASK)
               [TASK TSMFETCH (QUOTE CONCURRENT)
                     (NIL (EXECUTEMODEL (QUOTE CS.ANTECEDENT)
                                        (LIST TASK SUPERTASK]
               (CONCURRENT ANYTIME ((QUOTE START)
                                    (QUOTE *ILEQ)
                                    CHECKTIME)
                           (T (ALERT "OK!"))
                           (NIL (EXECUTEMODEL
                                  (QUOTE CS.ANTECEDENT)
                                  (LIST TASK SUPERTASK)))
                           (NODATA (QUEUE
                          "Missing START time for CONCURRENT tasks."
                                        (TASK SUPERTASK
                                              CONCURRENT)
                                        (PROCESSQUE]

[CS.ANTECEDENT (TASK NEXTTASK)
    (TASK EQ NIL (T (ALERT
              "FAIL.  TASK is NIL.  Can't go any further.")))
    (TASK TSMFETCH (QUOTE ANTECEDENT)
          (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                             NEXTTASK)))
    (ANTECEDENT ANYTIME ((QUOTE END)
                         (QUOTE *IGEQ)
                         CHECKTIME)
                (T (ALERT "FAIL"))
                (NIL (ALERT "OK!"))
                (NODATA (QUEUE
            "Missing END time for ANTECEDENT."
                        (TASK NEXTTASK ANTECEDENT)
                        (PROCESSQUE]
```

Appendix D

TRACES OF ACTUAL EXAMPLES OF TESTING UPDATE INFORMATION

Appendix D

TRACES OF ACTUAL EXAMPLES OF TESTING UPDATE INFORMATION

In this appendix we present traces of three examples of update-checking. The first one is accepted, the second one is refused, the final one is queued and delayed for further processing at a later date. Comments to explain what is going on are added in indented form.

Example 1:

In the first example, we check the update to the starting time of an aircraft servicing subtask, AC-SERV2. The indicated start time is 920.

```
_EXMODEL(AC-SERV2 920)
```

This is, in essence, the information that the update has occured. This function call to EXMODEL would normally be issued by the database management system.

```
EXECUTEMODEL: UPDATECHECK  (CHECKSTART AC AC1 1 AC-SERV2 920)
```

The update model of CHECKSTART is now used. The function EXECUTEMODEL uses that model to decide what to do by issuing SELECTQ's with arguments obtained from the update model. What we present now is the trace of those calls.

```
  (SELECTQ (EXECUTEMODEL CHECK NIL) NIL)

EXECUTEMODEL: CHECKSTART  NIL
```

One of the first operations to be performed is to get the planned start, end times and durations of this task. This information is obtained from the database via the DBFETCH function:

```
  (SELECTQ (DBFETCH CHECKTASK
                   (QUOTE (START END MIN.DUR MAX.DUR)))
           (NIL (PRIN1
    "Missing data in START, END, MIN.DUR or MAX.DUR.")
                (TERPRI))
           NIL)
```

```
DBFETCH: TASK=AC-SERV2
START: 930
END: 940
MIN.DUR: 10
MAX.DUR: 30
```

The update model indicates that the proposed start time should be within reasonable range of what was planned:

```
(SELECTQ (*INRANGE CHECKTIME (*IDIFF START 30)
                 (*IPLUS START 30))
         (NIL (ALERT
              "New start time not close to old start time."))
         (NODATA (QUEUE
              "Missing START" NIL NIL CHECKSTART NIL))
         NIL)
```

The update model indicates that the new implied duration should be close to what was planned:

```
(SELECTQ (*INRANGE (*IDIFF END CHECKTIME)
                 MIN.DUR MAX.DUR)
         (NIL (ALERT
        "New duration not within duration limits of TASK."))
         (NODATA (AND (NEQ START
                           (QUOTE NODATA))
                      (QUEUE
                       "Missing END, MIN.DUR or MAX.DUR"
                            NIL NIL CHECKSTART NIL))
                 (PROCESSQUE))
         NIL)
```

We must check that the updated time is greater than or equal to the planned start time. If yes, the update would be accepted; if no, we must check the supertask of AC-SERV2, i.e., the task that AC-SERV2 is part of. This is what happens in this case:

```
(SELECTQ (*IGEQ CHECKTIME START)
         (T (ALERT "OK!"))
         (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                            CHECKTASK))
         (NODATA (PROCESSQUE))
         NIL)
```

The update is tested with respect to the supertask of AC-SERV2:

```
EXECUTEMODEL: CS.SUPERTASK  AC-SERV2

(SELECTQ (EQ TASK NIL)
         (T (ALERT
             "FAIL.  TASK is NIL.  Can't go any further."))
         NIL)
```

First, we find the super-task from the time sequence model, as explained in Appendix A and B. This is done using the TSMFETCH function, which, in this case, answers that it is AC-SERV:

```
(SELECTQ (TSMFETCH TASK (QUOTE SUPERTASK))
         (NIL (ALERT "No SUPERTASK for TASK."))
         NIL)

TSMFETCH: (AC-SERV2 SUPERTASK AC-SERV)
```

We then get from the database the planned start time of this task:

```
(SELECTQ (DBFETCH SUPERTASK (QUOTE START))
         (NIL (QUEUE
                "No start time for SUPER.TASK"
                (TASK SUPERTASK)
                (PROCESSQUE)))
         NIL)

DBFETCH: TASK=AC-SERV
START: 920
```

We must now check that the updated time is greater than or equal to this start time--which is the case here:

```
(SELECTQ (*IGEQ CHECKTIME START)
         (NIL (EXECUTEMODEL (QUOTE CS.ANTECEDENT)
                            (LIST SUPERTASK SUPERTASK)))
         NIL)
```

Following that test, we must verify that the task that precedes AC-SERV2 was, or can be, satisfactorily completed. The function TSMFETCH is called recursively to find what is the antecedent of AC-SERV2 in AC-SERV--in this case, there is no such antecedent task, then the update is accepted:

```
(SELECTQ (TSMFETCH TASK (QUOTE ANTECEDENT))
         (NIL (ALERT "OK"))
         (T (EXECUTEMODEL (QUOTE CS.CONCURRENT)
                          (LIST TASK SUPERTASK)))
         NIL)

TSMFETCH: (AC-SERV2 SUPERTASK AC-SERV)
TSMFETCH: (AC-SERV SUBTASK AC-SERV2)
TSMFETCH: (AC-SERV2 ANTECEDENT NIL)
OK
"OK"
```

The update has been accepted.

Example 2:

The second example assumes the same update, but a different update checking model is assumed. The start of the update checking is the same as in the previous example.

```
_EXMODEL(AC-SERV2 920)

(CHECKSTART AC AC1 1 AC-SERV2 920)

EXECUTEMODEL: UPDATECHECK  (CHECKSTART AC AC1 1 AC-SERV2 920)

  (SELECTQ (EX
[SRI-KL going down in 30 minutes for 10 minutes.]
ECUTEMODEL CHECK NIL)
           NIL)

EXECUTEMODEL: CHECKSTART  NIL

  (SELECTQ (DBFETCH CHECKTASK
                    (QUOTE (START END MIN.DUR MAX.DUR)))
           (NIL (PRIN1
            "Missing data in START, END, MIN.DUR or MAX.DUR.")
                (TERPRI))
           NIL)

DBFETCH: TASK=AC-SERV2
START: 930
END: 940
MIN.DUR: 10
MAX.DUR: 30

  (SELECTQ (*INRANGE CHECKTIME (*IDIFF START 30)
                     (*IPLUS START 30))
           (NIL (ALERT
         "New start time not close to old start time."))
           (NODATA (QUEUE "Missing START"
                          NIL NIL CHECKSTART NIL))
           NIL)

  (SELECTQ (*INRANGE (*IDIFF END CHECKTIME)
                     MIN.DUR MAX.DUR)
           (NIL (ALERT
  "New duration not within duration limits of TASK."))
           (NODATA (AND (NEQ START (QUOTE NODATA))
                        (QUEUE
                         "Missing END, MIN.DUR or MAX.DUR"
                         NIL NIL CHECKSTART NIL))
                   (PROCESSQUE))
           NIL)
```

```
  (SELECTQ (*IGEQ CHECKTIME START)
           (T (ALERT "OK!"))
           (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                              CHECKTASK))
           (NODATA (PROCESSQUE))
           NIL)


EXECUTEMODEL: CS.SUPERTASK  AC-SERV2

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
      "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)

  (SELECTQ (TSMFETCH TASK (QUOTE SUPERTASK))
           (NIL (ALERT "No SUPERTASK for TASK."))
           NIL)

TSMFETCH: (AC-SERV2 SUPERTASK AC-SERV)

  (SELECTQ (DBFETCH SUPERTASK (QUOTE START))
           (NIL (QUEUE "No start time for SUPER.TASK"
                       (TASK SUPERTASK)
                       (PROCESSQUE)))
           NIL)

DBFETCH: TASK=AC-SERV
START: 930

  (SELECTQ (*IGEQ CHECKTIME START)
           (NIL (EXECUTEMODEL (QUOTE CS.ANTECEDENT)
                              (LIST SUPERTASK SUPERTASK)))
           NIL)
EXECUTEMODEL: CS.ANTECEDENT  (AC-SERV AC-SERV)

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
         "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)
```

The next check uses an update model that is different from the previous example. In this case, if AC-SERV2 does not have an antecedent in AC-SERV--which is the case--the user wants to test the update with respect to the supertask of AC-SERV, which in this case is POST-FLT:

```
  (SELECTQ (TSMFETCH TASK (QUOTE ANTECEDENT))
           (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                              NEXTTASK))
           NIL)
```

```
TSMFETCH: (AC-SERV SUPERTASK POST-FLT)
TSMFETCH: (POST-FLT SUBTASK ((AC-SERV PILOT-DEBRIEF)))
TSMFETCH: (AC-SERV ANTECEDENT NIL)
```

Tests with respect to a supertask are being executed. The supertask now is POST-FLT, but the tests are the same as those performed when the supertask was AC-SERV:

```
EXECUTEMODEL: CS.SUPERTASK  AC-SERV

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
     "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)

  (SELECTQ (TSMFETCH TASK (QUOTE SUPERTASK))
           (NIL (ALERT "No SUPERTASK for TASK."))
           NIL)

TSMFETCH: (AC-SERV SUPERTASK POST-FLT)

  (SELECTQ (DBFETCH SUPERTASK (QUOTE START))
           (NIL (QUEUE
                 "No start time for SUPER.TASK"
                 (TASK SUPERTASK)
                       (PROCESSQUE)))
           NIL)
```

The result is that the planned start time of *POST-FLT* is obtained:

```
DBFETCH: TASK=POST-FLT
START: 930
```

This is compared with the time appearing in the update. The updated time is earlier, requiring a test with respect to the antecedent of POST-FLT:

```
  (SELECTQ (*IGEC CHECKTIME START)
           (NIL (EXECUTEMODEL (QUOTE CS.ANTECEDENT)
                              (LIST SUPERTASK
                                    SUPERTASK)))
           NIL)
```

This causes the execution of the update model that tests the update time of a task with respect to an antecedent task:

```
EXECUTEMODEL: CS.ANTECEDENT  (POST-FLT POST-FLT)

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
         "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)
```

In this case, the antecedent to POST-FLT is found to be FLT:

```
  (SELECTQ (TSMFETCH TASK (QUOTE ANTECEDENT))
           (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                              NEXTTASK))
           NIL)

TSMFETCH: (POST-FLT SUPERTASK MISSION)
TSMFETCH: (MISSION SUBTASK (PRE-FLT FLT POST-FLT))
TSMFETCH: (POST-FLT ANTECEDENT FLT)
```

The end time of FLT is tested. It is required to be before the updated time--the start time of AC-SERV2. However, this is not the case, and the update is declined:

```
  (SELECTQ (ANYTIME ANTECEDENT (QUOTE END)
                    (QUOTE *IGEQ)
                    CHECKTIME)
           (T (ALERT "FAIL"))
           (NIL (ALERT "OK!"))
           (NODATA (QUEUE
         "Missing END time for ANTECEDENT."
                           (TASK NEXTTASK ANTECEDENT)
                           (PROCESSQUE)))
           NIL)

DBFETCH: TASK=FLT
END: 920
FAIL
"FAIL"
```

The update has been rejected.

Example 3.

The final example starts with the same update as the previous ones. The update model and the data retrieved from the database are initially also the same:

```
_EXMODEL(AC-SERV2 920)

(CHECKSTART AC AC1 1 AC-SERV2 920)

EXECUTEMODEL: UPDATECHECK
                (CHECKSTART AC AC1 1 AC-SERV2 920)

  (SELECTQ (EXECUTEMODEL CHECK NIL)
           NIL)

EXECUTEMODEL: CHECKSTART  NIL
```

```
(SELECTQ (DBFETCH CHECKTASK (QUOTE
      (START END MIN.DUR MAX.DUR)))
         (NIL (PRIN1
  "Missing data in START, END, MIN.DUR or MAX.DUR.")
              (TERPRI))
         NIL)

DBFETCH: TASK=AC-SERV2
START: 930
END: 940
MIN.DUR: 10
MAX.DUR: 30

  (SELECTQ (*INRANGE CHECKTIME (*IDIFF START 30)
                     (*IPLUS START 30))
           (NIL (ALERT
        "New start time not close to old start time."))
           (NODATA (QUEUE
        "Missing START" NIL NIL CHECKSTART NIL))
           NIL)

   (SELECTQ (*INRANGE (*IDIFF END CHECKTIME)
                      MIN.DUR MAX.DUR)
            (NIL (ALERT
     "New duration not within duration limits of TASK."))
            (NODATA (AND (NEQ START (QUOTE NODATA))
                         (QUEUE
                 "Missing END, MIN.DUR or MAX.DUR" NIL
                 NIL CHECKSTART NIL))
                    (PROCESSQUE))
            NIL)

   (SELECTQ (*IGEQ CHECKTIME START)
            (T (ALERT "OK!"))
            (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                               CHECKTASK))
            (NODATA (PROCESSQUE))
            NIL)
```

As in the previous examples we test the update with respect to AC-SERV, the supertask of AC-SERV2:

```
EXECUTEMODEL: CS.SUPERTASK  AC-SERV2

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
         "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)

  (SELECTQ (TSMFETCH TASK (QUOTE SUPERTASK))
           (NIL (ALERT "No SUPERTASK for TASK."))
           NIL)
TSMFETCH: (AC-SERV2 SUPERTASK AC-SERV)
```

```
  (SELECTQ (DBFETCH SUPERTASK (QUOTE START))
           (NIL (QUEUE "No start time for SUPER.TASK"
                       (TASK SUPERTASK)
                       (PROCESSQUE)))
           NIL)

DBFETCH: TASK=AC-SERV
START: 930

  (SELECTQ (*IGEQ CHECKTIME START)
           (NIL (EXECUTEMODEL (QUOTE CS.ANTECEDENT)
                              (LIST SUPERTASK
                                    SUPERTASK)))
           NIL)
EXECUTEMODEL: CS.ANTECEDENT  (AC-SERV AC-SERV)

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
          "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)

  (SELECTQ (TSMFETCH TASK (QUOTE ANTECEDENT))
           (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                              NEXTTASK))
           NIL)

TSMFETCH: (AC-SERV SUPERTASK POST-FLT)
TSMFETCH: (POST-FLT SUBTASK ((AC-SERV PILOT-DEBRIEF)))
TSMFETCH: (AC-SERV ANTECEDENT NIL)

EXECUTEMODEL: CS.SUPERTASK  AC-SERV

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
         "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)

  (SELECTQ (TSMFETCH TASK (QUOTE SUPERTASK))
           (NIL (ALERT "No SUPERTASK for TASK."))
           NIL)

TSMFETCH: (AC-SERV SUPERTASK POST-FLT)

  (SELECTQ (DBFETCH SUPERTASK (QUOTE START))
           (NIL (QUEUE "No start time for SUPER.TASK"
                       (TASK SUPERTASK)
                       (PROCESSQUE)))
           NIL)

DBFETCH: TASK=POST-FLT
START: 930
```

```
  (SELECTQ (*IGEQ CHECKTIME START)
           (NIL (EXECUTEMODEL (QUOTE CS.ANTECEDENT)
                              (LIST SUPERTASK
                                    SUPERTASK)))
           NIL)
```

As in the second example, the update is tested with respect to POST-FLT, the supertask of AC-SERV:

```
EXECUTEMODEL: CS.ANTECEDENT  (POST-FLT POST-FLT)

  (SELECTQ (EQ TASK NIL)
           (T (ALERT
         "FAIL.  TASK is NIL.  Can't go any further."))
           NIL)

  (SELECTQ (TSMFETCH TASK (QUOTE ANTECEDENT))
           (NIL (EXECUTEMODEL (QUOTE CS.SUPERTASK)
                              NEXTTASK))
           NIL)

TSMFETCH: (POST-FLT SUPERTASK MISSION)
TSMFETCH: (MISSION SUBTASK (PRE-FLT FLT POST-FLT))
TSMFETCH: (POST-FLT ANTECEDENT FLT)
```

The end time of FLT, the antecedent task of POST-FLT, is now retrieved. However, in contrast to the previous example, this time is not yet available:

```
  (SELECTQ (ANYTIME ANTECEDENT (QUOTE END)
                    (QUOTE *IGEQ)
                    CHECKTIME)
           (T (ALERT "FAIL"))
           (NIL (ALERT "OK!"))
           (NODATA (QUEUE
                    "Missing END time for ANTECEDENT."
                    (TASK NEXTTASK ANTECEDENT)
                    (PROCESSQUE)))
           NIL)

DBFETCH: TASK=FLT
END: NIL
Missing END time for ANTECEDENT.
```

The remaining part of the update model is therefore stored in the queue:

```
QUEUE: [33984 ((ANTECEDENT ANYTIME
                ((QUOTE END) (QUOTE *IGEQ) CHECKTIME)
                (T ( ALERT "FAIL"))
                (NIL (ALERT "OK!"))
                (NODATA
                   (QUEUE
        "Missing END time for ANTECEDENT."
```

```
                              (TASK NEXTTASK ANTECEDENT)
                              (PROCESSQUE)))))
                    ((TASK . POST-FLT)
                     (NEXTTASK . POST-FLT) (ANTECEDENT . FLT]
```

Later the queue is processed:

```
PROCESSQUE:

EXECUTEMODEL: ((ANTECEDENT ANYTIME
                           ((QUOTE END) (QUOTE *IGEQ)
                                        CHECKTIME)
                           (T ( ALERT FAIL))
                           (NIL (ALERT OK!))
                           (NODATA (QUEUE
                        "Missing END time for ANTECEDENT."
                                    (TASK NEXTTASK
                                          ANTECEDENT)
                                    (PROCESSQUE)))))
                ((TASK . POST-FLT) (NEXTTASK .  POST-FLT)
                                   (ANTECEDENT . FLT))
```

Again, an end time of flight is sought. This time one has been entered. The update can be satisfactorily completed:

```
  (SELECTQ (ANYTIME ANTECEDENT (QUOTE END)
                    (QUOTE *IGEQ)
                    CHECKTIME)
           (T (ALERT "FAIL"))
           (NIL (ALERT "OK!"))
           (NODATA (QUEUE
              "Missing END time for ANTECEDENT."
                           (TASK NEXTTASK ANTECEDENT)
                           (PROCESSQUE)))
           NIL)

DBFETCH: TASK=FLT
END: 915
OK!
"OK!"
```

The update has been accepted after recovery from the queue.

These three examples illustrate how update models are used to process inputted data.

DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street	1 copy
New York Area Office 715 Broadway - 5th Floor New York, New York 10003	1 copy
Naval Research Laboratory Technical Information Division Code 2627 Washington, D.C. 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps Code RD-1 Washington, D.C. 20380	1 copy
Office of Naval Research Code 455 Arlington, Virginia 22217	1 copy
Office of Naval Research Code 458 Arlington, Virginia 22217	1 copy
Naval Ocean Systems Center Advanced Software Technology Division Code 822 San Diego, California 92152	1 copy

(over)

Mr. E. H. Gleissner — 1 copy
Naval Ship Research & Dev. Center
Computation and Mathematics Dept.
Bethesda, Maryland 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS PLANNING BRANCH (OP-916D)
Office of Chief of Naval Operations
Washington, D.C. 20350

Director — 1 copy
National Security Agency
Attn: Mr. Glick
Fort George G. Meade, Maryland 20755

Naval Aviation Integrated Logistic Support Center — 1 copy
Code 800
Patuxent River, Maryland 20670

Professor Omar Wing — 1 copy
Columbia University in the City of New York
Department of Electrical Engineering and Computer Science
New York, New York 10027

Mr. M. Culpepper — 1 copy
Code 183
Naval Ship Research and Development Center
Bethesda, Maryland 20084

Mr. D. Jefferson — 1 copy
Code 188
Naval Ship Research and Development Center
Bethesda, Maryland 20084

Robert C. Kolb, Head — 1 copy
Code 824
Tactical Command Control and Navigation Division
Naval Ocean Systems Center
San Diego, California 92152

Defense Mapping Agency Topographic Center — 1 copy
Attn: Advanced Technology Division
Code 41300 (Mr. W. Mullison)
6500 Brookes Lane
Washington, D.C. 20315

(over)

Commander, Naval Sea Systems Command — 1 copy
Department of the NAvy
Attn: PMS 30611
Washington, D.C. 20362

Professor Mike Athans — 1 copy
MIT
Dept. of Elec. Eng. & Comp. Science
77 Massachusetts Avenue
Cambridge, Massachusetts 02139

Captain Richard L. Martin — 1 copy
Cmd. Officer, USS Francis Marion
LPA-249
FPO, New York 09051

Vivian Lee — 1 copy
DCASMA
1250 Bayhill Drive
San Bruno, CA 94066

Mr. Marvin Denicoff, Code 437 — 1 copy
Scientific Officer
Mathematical and Information Sciences Division
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Director, National Security Agency — 1 copy
ATTN: R53, Mr. Glick
Fort G. G. Meade, Maryland 20755

Contracts - 1copy of transmital letter

Barbara Haley " " "

Dee Leitner — 2 copies

Jack Goldberg — 1 copy

Dianne C-M — 1 copy

Brandin — 1 copy